IT & C

ISSN 2821 - 8469, ISSN – L 2821 - 8469, Volumul 2, Numărul 1, Martie 2023

Practici comune pentru programarea în C

Nicolae Sfetcu

Sfetcu, Nicolae (2023), Practici comune pentru programarea în C, *IT & C*, **2**:1, 30-36, DOI: 10.58679/IT80750, https://www.internetmobile.ro/practici-comune-pentru-programarea-in-c/

Publicat online: 09.02.2023

Practici comune pentru programarea în C

Nicolae Sfetcu
nicolae@sfetcu.com

Common Practices for C Programming

Abstract

With widespread use, a number of common practices and conventions have evolved to help avoid errors in C programming. These are both a demonstration of applying good software engineering principles to a language, and an indication of C's limitations. Although few are universally used and some are controversial, each enjoys wide use.

Keywords: programming, C programming language, C programming

Rezumat

Odată cu utilizarea pe scară largă, o serie de practici și convenții comune au evoluat pentru a ajuta la evitarea erorilor în programele C. Acestea sunt simultan o demonstrație a aplicării bunelor principii de inginerie software într-un limbaj, și o indicație a limitărilor C. Deși puține sunt utilizate universal, iar unele sunt controversate, fiecare dintre acestea se bucură de o utilizare largă.

Cuvinte cheie: programare, limbajul de programare C, programarea în C

IT & C, Volumul 2, Numărul 1, Martie 2023, pp. 30-36
ISSN 2821 - 8469, ISSN – L 2821 – 8469, DOI: 10.58679/IT80750
URL: https://www.internetmobile.ro/practici-comune-pentru-programarea-in-c/

Odată cu utilizarea pe scară largă, o serie de practici și convenții comune au evoluat pentru a ajuta la evitarea erorilor în programele C. Acestea sunt simultan o demonstrație a aplicării bunelor principii de inginerie software într-un limbaj, și o indicație a limitărilor C. Deși puține sunt utilizate universal, iar unele sunt controversate, fiecare dintre acestea se bucură de o utilizare largă.

Matrice multidimensionale dinamice

Deși matricele unidimensionale sunt ușor de creat dinamic folosind malloc, iar matricele multidimensionale cu dimensiuni fixe sunt ușor de creat folosind caracteristica de limbaj încorporată, matricele multidimensionale dinamice sunt mai complicate. Există o serie de moduri diferite de a le crea, fiecare cu compromisuri diferite. Cele mai populare două moduri de a le crea sunt:

- Ele pot fi alocate ca un singur bloc de memorie, la fel ca matricele multidimensionale statice. Acest lucru necesită ca matricea să fie dreptunghiulară (adică sub-tabele de dimensiuni mai mici sunt statice și au aceeași dimensiune). Dezavantajul este că sintaxa declarației pointerului este puțin complicată pentru programatorii începători. De exemplu, dacă s-ar dori să creeze o matrice de întregi de 3 coloane și rânduri `rows`, cineva ar scrie

```
int (*multi_array)[3] = malloc(rows * sizeof(int[3]));
```

(Rețineți că aici `multi_array` este un pointer către o matrice de 3 întregi.)

Din cauza interschimbabilității matrice-pointer, puteți indexa acest lucru la fel ca și matricele multidimensionale statice, de ex. `multi_array[5][2]` este elementul de pe al 6-lea rând și pe a 3-a coloană.

- Matricele multidimensionale dinamice pot fi alocate prin alocarea mai întâi a unei matrice de pointeri, apoi alocarea submatricelor și stocarea adreselor acestora în matricea de pointeri.[1] (Această abordare este cunoscută și ca vector Iliffe). Sintaxa pentru accesarea elementelor este aceeași ca și pentru tablourile multidimensionale descrise mai sus (chiar dacă sunt stocate foarte diferit). Această abordare are avantajul capacității de a realiza matrice neregulate (adică cu sub-matrice de diferite dimensiuni). Cu toate acestea, de asemenea, utilizează mai mult spațiu și necesită mai multe niveluri de direcționare indirectă pentru a indexa și poate avea performanțe mai slabe ale memoriei cache. De asemenea, necesită multe alocări dinamice, fiecare dintre acestea putând fi costisitoare.

Pentru mai multe informații, consultați întrebările frecvente comp.lang.c, întrebarea 6.16.

[1] Adam N. Rosenberg. [http://www.the-adam.com/adam/rantrave/st02.pdf "A Description of One Programmer's Programming Style Revisited"]. 2001. p. 19-20.

În unele cazuri, utilizarea tablourilor multidimensionale poate fi abordată cel mai bine ca o matrice de structuri. Înainte ca structurile de date definite de utilizator să fie disponibile, o tehnică comună a fost definirea unei matrice multidimensionale, în care fiecare coloană conține informații diferite despre rând. Această abordare este folosită frecvent și de programatorii începători. De exemplu, coloanele unei matrice de caractere bidimensionale pot conține nume, prenume, adresă etc.

În astfel de cazuri, este mai bine să definiți o structură care conține informațiile care au fost stocate în coloane și apoi să creați o matrice de pointeri către acea structură. Acest lucru este valabil mai ales atunci când numărul de puncte de date pentru o anumită înregistrare poate varia, cum ar fi melodiile dintr-un album. În aceste cazuri, este mai bine să creați o structură pentru album care să conțină informații despre album, împreună cu o matrice dinamică pentru lista de melodii de pe album. Apoi, o serie de pointeri către structura albumului poate fi folosită pentru a stoca colecția.

- Un alt mod util de a crea o matrice multidimensională dinamică este aplatizarea și indexarea manuală a matricei. De exemplu, o matrice bidimensională cu dimensiunile *x* și *y* are elemente x*y, prin urmare poate fi creată de

```
int dynamic_multi_array[x*y];
```

Indexul *x* este puțin mai complicat decât înainte, dar poate fi obținut totuși prin y*i+j. Apoi accesați matricea cu

```
static_multi_array[i][j];
dynamic_multi_array[y*i+j];
```

Mai multe exemple cu dimensiuni mai mari:

```
int dim1[w];
int dim2[w*x];
int dim3[w*x*y];
int dim4[w*x*y*z];

dim1[i]
dim2[w*j+i];
dim3[w*(x*i+j)+k] // index is k + w*j + w*x*i
dim4[w*(x*(y*i+j)+k)+l] // index is w*x*y*i + w*x*j + w*k + l
```

Rețineți că w*(x*(y*i+j)+k)+l este egal cu w*x*y*i + w*x*j + w*k + l, dar utilizează mai puține operații (metoda lui Horner). Folosește același număr de operații ca și accesarea unei matrice statice prin dim4[i][j][k][l], deci nu ar trebui să fie mai lent de utilizat.

Avantajul utilizării acestei metode este că matricea poate fi trecută liber între funcții fără a cunoaște dimensiunea matricei în momentul compilării (deoarece C o vede ca o matrice unidimensională, deși este încă necesară o modalitate de a trece dimensiunile), iar întreaga matrice este contiguă în memorie, deci accesarea elementelor consecutive ar trebui să fie rapidă. Dezavantajul este că poate fi dificil la început să te obișnuiești cu modul de indexare a elementelor.

Constructori și destructori

În majoritatea limbajelor orientate pe obiecte, obiectele nu pot fi create direct de către un client care dorește să le folosească. În schimb, clientul trebuie să ceară clasei să construiască o instanță a obiectului folosind o rutină specială numită constructor. Constructorii sunt importanți deoarece permit unui obiect să impună invarianți cu privire la starea sa internă pe toată durata de viață. Destructorii, chemați la sfârșitul duratei de viață a unui obiect, sunt importanți în sistemele în care un obiect deține acces exclusiv la o anumită resursă și este de dorit să se asigure că eliberează aceste resurse pentru a fi utilizate de către alte obiecte.

Deoarece C nu este un limbaj orientat pe obiecte, nu are suport încorporat pentru constructori sau destructori. Nu este neobișnuit ca clienții să aloce și să inițializeze în mod explicit înregistrări și alte obiecte. Cu toate acestea, acest lucru duce la un potențial de erori, deoarece operațiunile asupra obiectului pot eșua sau se pot comporta imprevizibil dacă obiectul nu este inițializat corespunzător. O abordare mai bună este să aveți o funcție care creează o instanță a obiectului, eventual luând parametrii de inițializare, ca în acest exemplu:

```
struct string {
    size_t size;
    char *data;
};

struct string *create_string(const char *initial) {
    assert (initial != NULL);
    struct string *new_string = malloc(sizeof(*new_string));
    if (new_string != NULL) {
        new_string->size = strlen(initial);
        new_string->data = strdup(initial);
    }
    return new_string;
}
```

În mod similar, dacă este lăsat la latitudinea clientului să distrugă obiectele corect, acesta poate să nu facă acest lucru, provocând scurgeri de resurse. Este mai bine să aveți un destructor explicit care este întotdeauna folosit, cum ar fi acesta:

```
void free_string(struct string *s) {
    assert (s != NULL);
    free(s->data);  ''/* free memory held by the structure */''
    free(s);        ''/* free the structure itself */''
}
```

Este adesea util să combinați destructorii cu indicatorii eliberați nuli.

Uneori este util să ascundeți definiția obiectului pentru a vă asigura că clientul nu îl alocă manual. Pentru a face acest lucru, structura este definită în fișierul sursă (sau într-un fișier antet privat care nu este disponibil utilizatorilor) în loc de fișierul antet și o declarație este pusă în fișierul antet:

```
struct string;
struct string *create_string(const char *initial);
void free_string(struct string *s);
```

Pointeri eliberați nuli

După cum s-a discutat mai devreme, după ce `free()` a fost apelat pe un pointer, acesta devine un pointer suspendat. Mai rău, majoritatea platformelor moderne nu pot detecta când este folosit un astfel de pointer înainte de a fi reatribuit.

O soluție simplă la aceasta este să vă asigurați că orice pointer este setat la un pointer nul imediat după ce a fost eliberat[2]:

```
free(p);
p = NULL;
```

Spre deosebire de pointerii suspendați, o excepție hardware va apărea pe multe arhitecturi moderne când un pointer nul este dereferențiat. De asemenea, programele pot include verificări de eroare pentru valoarea nulă, dar nu pentru o valoare a pointerului suspendat. Pentru a vă asigura că se face în toate locațiile, poate fi utilizată o macrocomandă:

```
#define FREE(p)   do { free(p); (p) = NULL; } while(0)
```

(Pentru a vedea de ce macro-ul este scris în acest fel, se pot folosi convențiile Macro.) De asemenea, atunci când se utilizează această tehnică, destructorii ar trebui să aducă la zero pointerul

[2] comp.lang.c FAQ list: "Why isn't a pointer null after calling free?" mentions that "it is often useful to set [pointer variables] to NULL immediately after freeing them".

pe care le-au trecut, iar argumentul lor trebuie să fie transmis prin referință pentru a permite acest lucru. De exemplu, iată destructorul de la *Constructori și destructori* actualizat:

```
void free_string(struct string **s) {
    assert(s != NULL  &&  *s != NULL);
    FREE((*s)->data);  ''/* free memory held by the structure */''
    FREE(*s);          ''/* free the structure itself */''
    *s=NULL;           ''/* zero the argument */''
}
```

Din păcate, acest idiom nu va face nimic altor pointeri care ar putea indica memoria eliberată. Din acest motiv, unii experți C consideră acest idiom ca fiind periculos din cauza creării unui fals sentiment de securitate.

Convenții macro

Deoarece macrocomenzile preprocesorului din C funcționează folosind înlocuirea simplă a simbolurilor, ele sunt predispuse la o serie de erori de confuzie, dintre care unele pot fi evitate urmând un set simplu de convenții:

1. Plasarea parantezelor în jurul argumentelor macro ori de câte ori este posibil. Acest lucru asigură că, dacă sunt expresii, ordinea operațiilor nu afectează comportamentul expresiei. De exemplu:
    a. Greșit: `#define square(x) x*x`
    b. Mai corect: `#define square(x) (x)*(x)`
2. Plasarea parantezelor în jurul întregii expresii dacă este o singură expresie. Din nou, acest lucru evită schimbările de sens datorită ordinii operațiilor.
    a. Greșit: `#define square(x) (x)*(x)`
    b. Mai corect: `#define square(x) ((x)*(x))`
    c. Periculos, amintiți-vă că înlocuiește cuvintele din text. Să presupunem că codul tău este `square (x++)`, după invocarea macro-ului *x* va fi incrementat cu 2
3. Dacă o macrocomandă produce mai multe instrucțiuni sau declară variabile, aceasta poate fi înfășurată într-o buclă **do** { ... } **while**(0), fără punct și virgulă de sfârșit. Acest lucru permite macrocomenzii să fie folosită ca o singură instrucțiune în orice locație, cum ar fi corpul unei instrucțiuni `if`, permițând totuși plasarea punctului și virgulă după invocarea macrocomenzii fără a crea o instrucțiune nulă[3][4][5][6][7][8]. Trebuie avut grijă ca orice variabilă nouă să nu mascheze potențial porțiuni din argumentele macrocomenzii.
    a. Greșit: `#define FREE(p) free(p); p = NULL;`
    b. Mai corect:: `#define FREE(p) do { free(p); p = NULL; } while(0)`

---

[3] "comp.lang.c FAQ: What's the best way to write a multi-statement macro?".
[4] "The C Preprocessor: Swallowing the Semicolon"
[5] "Why use apparently meaningless do-while and if-else statements in macros?"
[6] "do {...} while (0) in macros"
[7] "KernelNewbies: FAQ / DoWhile0".
[8] "PRE10-C. Wrap multistatement macros in a do-while loop".

4. Evitarea folosirii unui argument macro de două sau mai multe ori în interiorul unei macrocomenzi, dacă este posibil; acest lucru cauzează probleme cu argumentele macro care conțin efecte secundare, cum ar fi atribuirile.
5. Dacă o macrocomandă poate fi înlocuită cu o funcție în viitor, luați în considerare denumirea acesteia ca o funcție.
6. Prin convenție, valorile preprocesorului și macrocomenzile definite de `#define` sunt denumite cu toate literele mari[9][10][11][12][13].
   Există un număr mare de instrucțiuni pentru stilul C.

- „Ghidurile de stil C și C++” de Chris Lott enumeră multe ghiduri de stil C populare.
- Motor Industry Reliability Association (MISRA) publică „MISRA-C: Ghid pentru utilizarea limbajului C în sistemele critice”.

Bibliografie

Adam N. Rosenberg. [http://www.the-adam.com/adam/rantrave/st02.pdf "A Description of One Programmer’s Programming Style Revisited"]. 2001. p. 19-20.
comp.lang.c FAQ list: "Why isn't a pointer null after calling free?" mentions that "it is often useful to set [pointer variables] to NULL immediately after freeing them".
"comp.lang.c FAQ: What's the best way to write a multi-statement macro?".
"The C Preprocessor: Swallowing the Semicolon"
"Why use apparently meaningless do-while and if-else statements in macros?"
"do {...} while (0) in macros"
"KernelNewbies: FAQ / DoWhile0".
"PRE10-C. Wrap multistatement macros in a do-while loop".
"What is the history for naming constants in all uppercase?"
"The Preprocessor".
"C Language Style Guide".
"non capitalized macros are always evil".
"Exploiting the Preprocessor for Fun and Profit".

[9] "What is the history for naming constants in all uppercase?"
[10] "The Preprocessor".
[11] "C Language Style Guide"
[12] "non capitalized macros are always evil"
[13] "Exploiting the Preprocessor for Fun and Profit"